SMART: Security Measurements and Assuring Reliability through metrics Technology

Final Report
November 2009
Principal Investigators: Wayne Zage and Dolores Zage
Sponsor: U.S. Army Research Laboratory

1. Introduction

Battlefield operations in the foreseeable future will depend heavily on network-centric computing systems that link a diverse multitude of geographically dispersed resources, operating on widely varied platforms, into a cohesive fighting force. The war fighter at all levels will depend on these unified systems to conduct successful multi-force operations in the 4-dimensional battle space. Such complex and widely dispersed operations expose network-based systems to unprecedented levels of reliability and security risks.

Computer systems and network security are often limited by the reliability of the software running on constituent machines. Faults in the software expose vulnerabilities, pointing to the fact that a critical aspect of the computer security problem resides in software. Security holes and vulnerabilities are often the result of bad software design and implementation. Since reliability and security are so closely intertwined, this research focused on analyzing the reliability and security of a system. Being able to assess the security and reliability of the software is essential to the overall mission of the United States military.

This research proposed to extend the principal investigators' proven metrics technology, combined with their extensive technical resources, to address the theoretical and technological underpinnings of widely dispersed network-centric software component design. The goal of this research was to provide component-design level information to support the accurate prediction of the reliability and security of individual and interdependent components in a network-centric environment. Successful prediction involves two levels of system understanding, architectural risk analysis and implementation analysis. Combining both analyses provided a higher likelihood of success.

2. Background

2.1 Design Metrics

The design metrics research team at Ball State University has developed a metrics approach for analyzing software designs that helps designers engineer quality into the design product. Two of the design metrics developed are an external design metric D_e and an internal design metric D_i. The calculation of D_e focuses on a module's *external* relationships to other modules in the software system and is based on information available during architectural design. The metric D_i incorporates factors related to a module's *internal* structure and is calculated during the detailed design phase of software development. These metrics gauge project quality as well as design complexity during the design phase and subsequent phases thereafter [ZAGE90]. The primary objective is to employ these metrics to identify troublesome modules, or stress points, in the structure of the software.

To corroborate the effectiveness of the metrics as predictors of fault-proneness, the design metrics team reexamines the software when error reports are available to determine to what extent the stress-point modules were indeed fault-prone. Over an extensive metrics evaluation and validation period, on study data consisting of university-based projects and large-scale industrial software, these design metrics consistently outperformed classical, well-known metrics such as LOC, V(G), and information flow metrics, as predictors of fault-prone modules [ZAGE93].

The design metrics D_e and D_i have been computed on a number of systems and subsystems obtained from our industrial collaborators, including

- Computer Sciences Corporation Standard Financial System (STANFINS)
- Systems from the US Army Research Lab
- Raytheon's Advanced Field Artillery Tactical Data System (AFATDS)
- Harris' Radar Online Command and Control (ROCC) system
- Three Northrop Grumman projects
- A PBX system from Telcordia Technologies
- Telecommunications systems from Motorola

The empirical results over the entire validation period can be summarized by stating that the design metrics have identified at least 76% of the detected fault-prone modules 100% of the time.

The design metrics technology also has been applied to an SDL telecommunications environment. SDL is a standard language for the specification and description of systems. SDL is used in the telecommunications area, as well as in other real-time, distributed and communicating systems to increase productivity, reduce errors and improve maintainability. Mappings from SDL to code have been developed [FLOC95]. One of our initial objectives was to map SDL artifacts to design metrics primitives. In our SDL research, we conducted a metrics analysis of SDL designs and assessed the utility and effectiveness of our design metrics on such models [ZAGE99]. We also refined our metrics model to incorporate additional SDL features, analyzed extensive error reports and observed metrics patterns that correlated with high error density SDL components [WONG00].

There have been several attempts at extracting artifacts from software designs and calculating metrics from them or comparing them to established patterns. Some of these attempts have focused on the Unified Modeling Language (UML), since it has become the de facto graphical language for specifying, constructing, visualizing, and documenting software systems. These efforts have been an attempt to gauge the quality of the design early in the lifecycle when it is economically beneficial to make changes. It is well known that correcting an error encountered by an end-user is an order of magnitude more expensive than when finding it in earlier phases [ZAGE03]. Researchers are focused on two approaches for analyzing the quality of the design; comparing the existing design to existing design patterns or using metrics to identify risky areas in the design [ZAGE99].

In a recent study, we asked if UML classes with high numbers of change orders were highlighted as stress points by D_e. The results indicated that D_e correctly classified UML classes 89% of the time, with the conclusion that D_e can accurately identify the most problematic UML classes, as given by change order values [ZAGE06].

2.2 Software Reliability

A significant amount of software reliability research has focused on software reliability estimation models. Existing methods for reliability estimation use program failure data or repeated appearance of the faults as inputs to an estimation model to predict the reliability of the software [MUSA87, CHAO93, YANG95]. It has been argued that existing methods for software reliability estimation may fail to provide accurate reliability estimates [HORG95]. This may happen, for example, in the presence of an inaccurate operational profile. Experiments carried out using simulations and "real" programs provided evidence in support of the above argument [CHEN94a, CHEN94b]. A natural question to ask then is "how can one obtain reasonably accurate estimates of software reliability?"

Experimental work provides evidence in support of the hypothesis that measurable attributes of software and of software designs are related to the reliability of the fielded code [STIN05]. The results of this study indicate that a positive relationship exists between the design metric D(G), which is a linear combination of D_e and D_i, and the number of defects found in the product after it has been fielded. The relationship has been identified and quantified to a first approximation. Further, it has been shown that D(G) can be used to predict and measure the impact of design decisions on the expected reliability of a software product during the code and tests phases of development. The ability to identify and measure stress points in the software has also been shown. The concept of design significance was introduced and shown to be a viable and measurable attribute that can identify those components for which design decisions have a significant effect on the expected reliability of the software. These findings were sufficient to justify development and implementation of a high reliability engineering process in a Software Engineering Research Center (SERC) affiliate company.

2.3 Software Security

Software designs can be evaluated for software protection by applying three complementary techniques. First, a vulnerability assessment can be performed by calculating and interpreting metrics on the designs to identify, categorize and analyze security weaknesses. A second technique is to measure the conformance of the design to a specific style or pattern. Finally, the design can be scrutinized for detection of anti-patterns (patterns to be avoided).

Intuitively, one would expect that vulnerability is directly proportional to complexity. Complex systems are often fielded with major, undetected vulnerabilities. Even with improved verification, laboratory testing and user testing, security vulnerabilities are fielded in delivered systems. According to most researchers and practitioners, the most common category of security vulnerabilities is flaws in software. Many of the observed problems are a result of poor design, but there are other issues that cause security problems as well.

It is important for modern system developments to integrate anti-tamper technologies in order to protect critical program information. This does not come without cost, however, and must be carefully planned, developed, and tested to achieve maximum effectiveness. The principal investigators, in collaboration with researchers at Arxan Technologies, have explored cost models for determining the cost of anti-tamper security deployments in software systems [BRYA05]. The categories of protection design considered were manual, semi-automated, and fully-automated. These three categories are not only major cost drivers for the protection, but also have a major impact on its strength.

2.4 Security Metrics

Much work has been done in the area of identifying potential vulnerability-enabling code. Tools such as ITS4, RATS and FlawFinder identify specific functions within code. However, for security evaluation tools, these types of function-identifying tools are only marginally useful [HEFF04]. Many of these tools are simply function pattern matchers and will not provide any useful barometer to future attacks.

Michael Howard of Microsoft first introduced an attack surface metric for the Windows operating system [HOWA03]. Since then it has been extended by several researchers [MANA05, GOLU05]. The basis of this metric is the opportunities of attack measured by notion of an attack surface of the system. The limitations of this metric are that it is specific to one system and cannot handle configuration changes.

For the creation of the design metrics, the question that focused the search for the selection of the primitives was "Where do software designers or programmers commonly make mistakes within the development?" For security metrics, the question is, "How do external threats (attacks) enter the system?" An additional question is "What are the prerequisites of an attack?" Some researchers have included the concept of entry and exit points. The current external design metric D_e also includes measurements of these concepts.

2.5 Modular Assessment Process

As modules can be composed of collections of other modules, multiple module measurement iterations will contribute to the overall system measurement. The result is a set of reliability and security measurements and information that can support operational acceptance or structural integrity of the individual components as they are combined. The assessment process mimics software developed through components or component-based software development (CBSD). Similarly, reliability and security measurement has to be considered over an enormous range of types of technology and components. For example:

- Software at code level, bit/register level
- Software module, object
- Software application
- Software system, architecture
- System, aggregation of software and hardware components (single, monolithic entity)
- Networked system, where communication links and nodes lie within protected environments
- System of systems, i.e., systems that are developed to independent goals, but are required to inter-operate
- Systems with specific prime function; information processing, command & control, embedded real-time control etc.
- System or component with a prime function to mitigate security risk
- Internet technology component or system, where communication links and nodes are provided by many other parties
- Grid systems
- Mobile/ubiquitous systems [MURD06]

For systems of the types listed above, the metrics must support aggregated levels of systems and services. However, the properties of compositions have not yet been defined. Systems are bound to change and that will result in additions or removals of components. To handle these possibilities, the measurements of security and reliability properties also should be amenable to change.

2.6 Visualization

Applying visualization to software has the potential to rapidly isolate the reliability and susceptibility problems, quickly leading to significant program improvements. Although the motivation for designing visualization tools may be obvious, how to design an effective tool is not so obvious. Many visualization tools exist but few of them are widely used in practice. Visualizing and understanding complex software architecture requires an enormous amount of effort and concentration. To assist in this task, an extensive amount of work has been done to develop visual browsers that provide the user with hierarchical views of the files, classes, and calls. Although these visual browsers exist, they are not widely used and have not been included in current development processes [LANZ03]. Traditionally, visualization tools present extractable facts as graphs or charts, but do not have as their goal an analysis and interaction emphasis. This lack of emphasis on analysis and interaction produces visualizations that are all content, but are lacking context, or meaning. The comprehensive assessment proposed in this research requires a rich and plentiful set of data. To understand this information, a visual environment replicating dimensions and layers is required to present the relevant information and characteristics of the system. After the specification of the framework was completed, a Reliability Appraisal and Vulnerability Evaluation (RAVE) environment was designed. A set of heuristics was compiled to determine the selection of appropriate visualizations (box trees, tree and file maps, graphs and layouts, etc.).

3. Areas of Focus

- Reliability assessment
- Vulnerability assessment
- Metrics
- Unified assessment process
- Visualization and computerization of a unified assessment process

4. Objective

Our overall objective of the SMART project was to create a unified reliability and security assessment process for software systems. To meet the overall objective of creating such a process the following sub-objectives must be met:

Directly measure software reliability by quantifying component level design and implementation information that accurately predicts the reliability of individual and interdependent components.

Directly measure software security by quantifying component level design and implementation information that accurately predicts the security of individual and interdependent components.

Design and implement interface visualization tools for reliability and security assessments (RAVE prototype).

5. Challenges

The quantitative measurement of software reliability and security has been a long-standing challenge. As stated in the 2002 Computer Research Association Conference on "Grand Research Challenges" in Computer Science and Engineering, one grand challenge is to create systems "you can count on". This means that applications must be reliable and secure to enable a whole new class of critical services. The new systems will contain a large number of heterogeneous units that evolve, accommodate change and grow. This challenge is just not technology related, but is a critical national goal that depends upon and drives technology. Outlined from this conference were ten technical challenges to achieve the goal of systems you can depend on. Three of these are directly pertinent to this proposal.

- Develop meaningful metrics of system security, stability, etc.
- Develop system auditing and analysis techniques
- Develop broad architectural rethinking

To develop meaningful metrics, one of the challenges is not only to identify current vulnerabilities, but also to highlight future vulnerabilities. Since 1988, the Computer Emergency Response Team Coordination Center (CERT/CC) at Carnegie Mellon University has been tracking security incidents. The number of security incidents has been doubling every year. Even more significantly, the number of distinct vulnerabilities is also growing at an exponential rate and these statistics are only for the vulnerabilities reported. Even though existing tools have enabled developers to quickly identify suspicious signatures, the unpredictability of new attacks requires a much broader approach.

Another challenge is the disappearance of a useful notion of a defensive "perimeter". Threats can stem from anywhere and attempting to exclude attackers is no longer an option. Closely related to seamless perimeters are composability challenges. For example, in military coalition operations, new ad hoc partnership structures will be created dynamically. These partnerships are necessary for flexible communication and coordination and can be created by adding new components to an already dynamic system. Rules guaranteeing that two systems operating together will not introduce vulnerabilities that neither have individually are either nonexistent or still in their infancy. Systems can also be constructed with many technologies and a single assessment technique may not be enough to handle their diversity.

The complete assessment system must produce a synergistic evaluation where redundant detections must be suppressed, complementary detections combined and localization of component assessment along with system assessment must be available as well as accurate mappings to source. Therefore, a unified assessment of reliability and security also is a challenge.

The creation of a user-friendly visualization tool is difficult and time-consuming; it depends heavily on the choice of visual metaphors and interface design. The metrics that will have the highest impact upon security and reliability are not known a priori, and so the design of the visualization tool is dependant on progress on the analytical front. The completion of the visualization tool will be hindered by delays or refactoring in any other part of the process, such as data acquisition, data analysis, metric selection and testing. We hypothesize that the visualizations can be based on UML and other common graphical languages. However, we will not force the data to fit into these models if it is inappropriate. It may be necessary to develop custom notations or adopt uncommon visual languages to best fit the security and reliability metrics. This will likely increase the amount of training required to use the tool, compared to cases where conventional notations are applicable.

Meeting any of the challenges listed here will extend the boundaries of quantifying software reliability and security.

6. Approach

The objective of this research was a unified reliability and security assessment process. We believe that reliability and security are inexplicably linked (see Figure 1). Most vulnerabilities are unintentional, not all defects are exploitable vulnerabilities, and not all vulnerabilities are defects. These defects can be *accidental*, or they can be *intentional*, and the latter may be malicious (Trojan horses, mtrap-doors) or non-malicious (resulting, for example, from deliberate trade-offs between security and efficiency). All appear to have reliability analogies except malicious intentional defects. From previous work, a software reliability metrics framework had been developed. This framework was evaluated for its vulnerability-discovery rate. Several systems with data on previously discovered vulnerabilities were studied to examine the dynamics of vulnerability discovery.

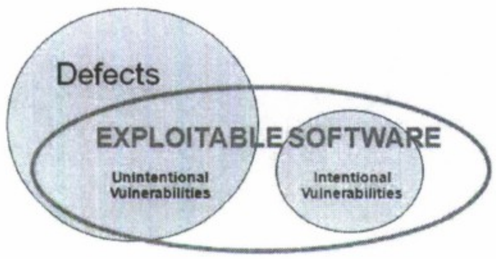


Figure 1: Defects and Vulnerabilities

In the proposed work, we are extending the latest research in software reliability and modeling to address complex network-centric systems. Analysis began with the external design metric, D_e, which provides a reliability estimate at the component design level, and the internal design metric, D_i, which provides a reliability estimate at the code level. As we investigate enhancements to the design metrics related to security, the metric D_i will be augmented to account for the kinds of low-level attacks that can be made in C, C++ and Java. As depicted in Figure 2, we see a definite parallel of our reliability process and the new security process. The most common cause of adversarial-initiated harm is exploitation of programming defects and these vulnerabilities overlap with fault-prone modules. Security holes and vulnerabilities are often the result of bad software design and implementation and since the design metrics are good at identifying fault-prone components, we believe we have a good start at identifying vulnerabilities in software.

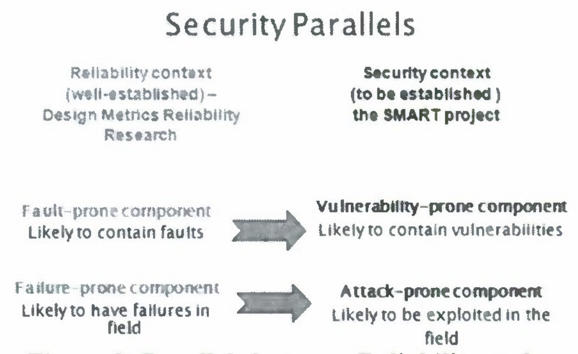

Security Parallels
Reliability context (well-established) – Design Metrics Reliability Research
Security context (to be established) the SMART project
Fault-prone component
Likely to contain faults
Vulnerability-prone component
Likely to contain vulnerabilities
Failure-prone component
Likely to have failures in field
Attack-prone component
Likely to be exploited in the field

Figure 2: Parallels between Reliability and Security in Software

Sometimes software security engineers are given a product that they not familiar with and are asked to do a security analysis of it in a relatively short time. A knowledge of where vulnerabilities are most likely to reside can help prioritize their efforts. In general, software metrics can be used to predict fault- and failure-prone components for prioritizing inspection, testing, and redesign efforts. We believe that the security community can leverage this knowledge to design tools and metrics that can identify vulnerability and attack-prone components early in the software life cycle. We analyzed a large commercial telecommunications software-based system and found that the presence of security faults correlated strongly with the presence of a more general category of reliability faults. This, of course, is not surprising if one accepts the notion that security faults are in many instances a subset of a reliability fault set.

However, security is more complicated and is a system-wide property where real attackers are actively trying to compromise software. The overall architecture which results from a combination of OS-level and third-party components needs to be reviewed and analyzed for the uncovering of security problems. Most applications are designed to span multiple boundaries of trust, and the vulnerability of any given component varies with the platform (e.g., J2EE application on Tomcat/Apache/Linux) and with the environment (client network versus Internet). Security must be assessed from two levels, namely the component level and the environment. Few security assessment frameworks address the variability of the core environment and this could be fatal when considering highly distributed applications, service oriented architectures or web services. Our assessment of the component-by-component basis (internal and external) is being evaluated through the original design metrics. Additionally, the environmental issues (language, operating system, container systems, authentication methods, network, etc.) will be measured through a new set of environment metrics which we call E_e and E_i (external and internal). Whereas the metrics D_e and D_i assess the external and internal design complexity of a module using primitives such as inflows, outflows, fan-in, fan-out, data structure manipulations and central calls, the external environment metric E_e assesses the environment level and the internal environment metric E_i assesses the local architectural impact, allowing for two additional levels of system understanding. (See Figures 3 and 4.)

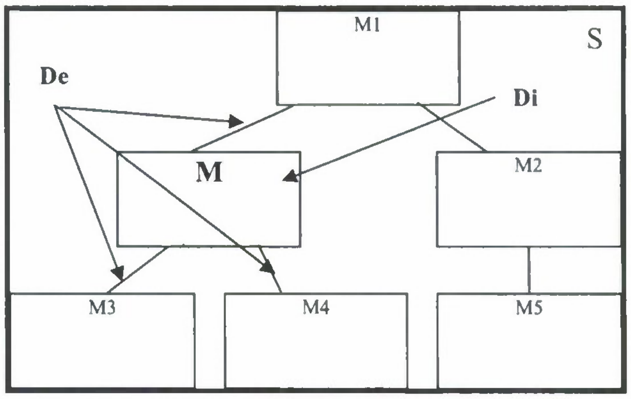


Figure 3: Locations of Primitive Values for D_e and D_i for a module M in a system S

The attempt to quantify an objective indication of potential software reliability and security is a major step in the process of improving current systems. Our proposed quantification takes an aggregate snapshot approach instead of addressing a particular aspect of risk. This research approaches risk at two points, reliability and vulnerability. Additionally, reliability and vulnerability should be addressed at the various layers of the system starting with the individual system components and reaching to the higher layers of the platform and network. Each layer will be individually assessed and integrated to provide a robust and accurate picture of a system's reliability and security posture.

To complete the layered approach, the metric primitives that comprise the environmental metrics E_e and E_i must be investigated and determined. The following items are being considered during the development and analysis of metrics for the purpose of measuring a software component or architecture on the basis of its security risk:

- Metrics must yield quantifiable information (percentages, averages, etc.)
- Data supporting metrics need to be readily obtainable
- Only repeatable processes will be considered for measurement
- Metrics must be useful for tracking performance and directing resources.

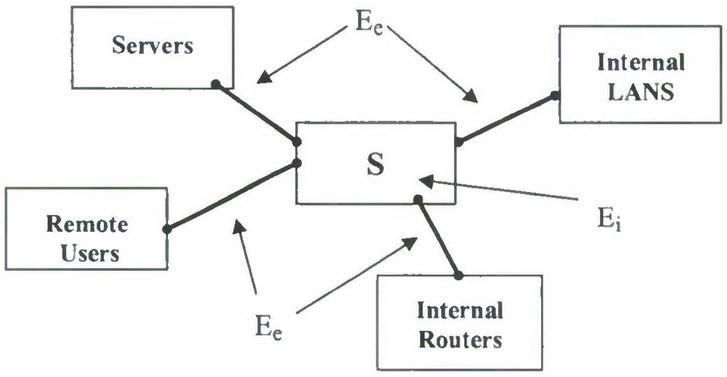


Figure 4: Locations of Primitive Values for E_e and E_i for the System S Shown in Figure 3

A comprehensible, cohesive picture of a system's reliability/security assessment requires an integrated knowledge solicitation consolidated into a topological framework. This framework should encompass the network topology, each individual system's architecture, threat data and environment data in order to link system level concerns to the design of the software. The framework must support our analysis for a module, a component, a tier, and an environment layer while also applying the different principles of measuring threats and vulnerabilities for the respective layers.

6.1 SMART Visualization Concept

This section describes the motivation behind *mediated metrics*, metrics whose values are specified interactively by a security metrics engineer. Incorporating this form of metric analysis requires tool support, and a conceptualization of such a tool, *RAVE*, is presented.

Best practices dictate that large software systems should be developed in *modules*. By separating concerns into modules, a software engineer can reduce the coupling and cohesion between subsystems. This has many benefits including the facilitation of communication and maintenance. However, the definition of module boundaries is sometimes subjective, especially in large software systems where in a module may comprise many smaller modules. Component-based systems, service-oriented architectures, and all object-oriented software are conceptually composed of many hierarchies of modules.

The Zage design metrics, D_e and D_i, combine intra- and inter-module measurements to gauge the complexity and error-proneness of software systems [ZAGE93]. This module-based approach to software analysis naturally extends to security and reliability research, where intra- and inter-module metrics can reveal, during design, software's proneness towards vulnerability.

There are two primary complications to predicting vulnerabilities using metrics:

1. the hierarchical and module boundaries may vary from system to system, and
2. the relative impact of potential vulnerabilities may be difficult or impossible to determine automatically.

A brief description of each of these complications is provided below, followed by an explanation of the role of visualization in ameliorating these difficulties.

6.1.1 Complications of Module-based Metric Analysis

The acquisition of module boundary information requires strict definitions that are dependent on implementation language, context, and environment. Measuring software after it is implemented is complicated by the separation of the original design from the actual implementation. There is strong potential for incremental or even fundamental changes in software design from its conceptualization to its implementation. The module boundary definitions that one may extract from software artifacts, such as directories or namespaces, may or may not reflect the semantic model of the software designer.

Module definitions are frequently hierarchical. The libraries, packages, components, and types that make up a program are nearly always organized within a nested hierarchy. Depending on one's perspective, each level may be considered a "module." The Java API provides a convenient example of this phenomenon. The package javax.swing contains (most of) the Swing user-interface library, and so it may be considered to be a module. However, within this package are nested packages javax.swing.table, javax.swing.plaf, *etc.*, and when one is considering the design of the Swing API, each of these is a module.

Regarding the second complication of vulnerability prediction through metrics analysis, there are categories of vulnerability that are difficult or impossible to correctly analyze automatically. While metrics, such as inflows and outflows can be determined through static analysis (though not without their own complications), other classifications do not share this luxury. For example, the degree to which a module is shared across threads and processors is important when assessing potential vulnerabilities. Determining whether or not a method executes concurrently is undecidable. A similar problem exists for I/O vulnerabilities. Network I/O has higher potential for security vulnerabilities than local storage I/O, but in languages such as C, both are abstracted by integer file descriptors after the device connection is opened. Hence it may be difficult or impossible, especially in the context of method pointers or dynamic code generation, to determine exactly what kind of I/O is being performed for any primitive read or write operation.

6.1.2 The Role of Visualization

The complications of security analysis through module-based metrics can be minimized through *mediated analysis*. Rather than attempting to automatically collect metric data from software design artifacts, whether at design time or afterwards, a designer can use a software tool to interactively mediate the analysis. An obvious drawback to such an approach is that a security metrics engineer would require training in both security metrics and the tool itself, but the potential advantages are worthwhile. Such a tool could be used during early design, software implementation, and post-mortem analysis to locate modules with potential security vulnerabilities.

Measurements that are difficult to collect automatically can be directly manipulated by a security metrics engineer. As examples, we revisit level of concurrency and I/O classification. A software designer should be able to estimate the relative concurrency of modules. That is, one who understands the design of a system should be able to differentiate between modules that tend towards high levels of concurrency and those that are only run on one or few threads. The reliability or security problems associated with concurrency, such as deadlock or race conditions, can be given appropriate weights on a per-module basis. Similarly, a security metrics engineer can interactively assign appropriate weights to modules based on an understanding of their I/O requirements.

Hierarchical modules can be effectively visualized using established visualization techniques. Nested module designs can be represented using hierarchy graphs, which can be interactively visualized using techniques such as those described by Buchsbaum and Westbrook [BUCH00]. For very large systems, where the module hierarchy is too complex for a single user to navigate, visual hierarchical aggregation can be used to present the user with a graph of reduced complexity that still expresses the essential properties of the design [NOEL04]. This strategy has been effectively used for visualizing complex attack graphs [SHEY02].

6.2 Technology Considerations

Java was selected as the primary implementation language for this project. Static type checking is desirable for a large-scale, extensible, modular system. The capacity for rapid prototyping is reduced, but this tradeoff is a net positive: extensibility and robustness are more appropriate goals for this project.

Eclipse (http://www.eclipse.org) will be used as a development environment. It provides a useful interface to the standard tools expected of a modern IDE, and support for subversion (http://subversion.tigris.org) can be added through the Subclipse plug-in (http://subclipse.tigris.org). These are all free, open-source tools, and their use does not encumber any software licensing restrictions.

There are several options available for the module drawing engine. Popular utilities for graph drawing in Java include GEF (http://www.eclipse.org/gef), JGraph (http://www.jgraph.org), and Java2D (http://java.sun.com/products/java-media/2D). These libraries are well-known and relatively simple, but they do not elegantly support animation or zoomable user-interfaces (ZUIs). Java ZUI libraries include Piccolo (http://www.cs.umd.edu/hcil/jazz), prefuse (http://www.prefuse.org), and ZVTM (http://zvtm.sourceforge.net). Of these, prefuse is the most robust and most complicated, in analogy to OpenGL for 3D graphics: it is low-level enough to allow practically anything to be built on top of it, but because it is a low-level library, development times are increased. It is likely that prefuse is the best tool available, but since the visual metaphors for mediated module are not well understood, it is premature to tie visualization to a specific graphics engine at this time. The proposed solution is to build an abstraction layer for the visualization, which will allow for quick proofs-of-concept to be generated in a relatively simple, high-level library such as Swing, and once the user interactions are better understood, this can be swapped with a prefuse implementation (or another, if it is deemed more appropriate).

As mentioned above, abstraction and extensibility are key to this project. Wherever possible, human-readable and verifiable data formats such as validated XML will be used, since these will facilitate development of modules regardless of their dependencies' being complete. For example, an engineer could craft an XML document describing a system's values for some metric *x*, even if the associated metric tool did not yet support the collection and analysis of *x*.

7. SMART Project

7.1 Study Data Characterization

To begin collecting data regarding vulnerabilities, a list of requirements for software selection were determined. First, the software had to be mature to ensure that there was a long list of vulnerabilities. Second, the vulnerabilities needed to be well-documented. Third, the vulnerabilities needed to span a wide variety of categories so that the under-lying design issues across different vulnerability types could be understood. Finally, the source code for the projects needed to be available so that the vulnerabilities could be located in the code. We selected open source systems that are typical in a network environment to broaden the type of applications under consideration.
The study data include:

- Apache http server –
    - Since April 1996 Apache has been the most popular HTTP server on the WWW.
    - As of March 2009 Apache served over 46% of all websites and over 66% of the million busiest.
- OpenSolaris – an operating system
- FireFox –
    - Firefox had 22.05% of the recorded usage share of web browsers as of March 2009, making it the second most popular browser in terms of current use worldwide, after Internet Explorer.
- OpenSSH - Set of computer programs providing encrypted communication sessions over a computer network using the ssh protocol.
- Drupal - Content Management System (CMS)
    - Over 350,000 subscribed members
    - DrupalSites.net is a directory that list thousands of websites powered by Drupal
    - Winner of Best Overall 2008 Open Source CMS Award for Second Year in a Row
    - Listed as one of the Open Source PHP applications that changed the world

All of the open source systems were selected for analysis possessed multiple versions. The class diagram and call graph for each selected system was generated. Each system is broken down to the component level. For each component, a set of metrics was collected and for each problem report, components changed to correct the problem were identified. Problems were classified by severity (catastrophic, severe, major, minor, no effect), defect type, and classification of software vulnerability.

One of the initial steps of the project was to survey the different types of software vulnerabilities and place them into specific categories. During the survey process, the Common Weaknesses and Enumeration (CWE), a community-developed dictionary of common software weaknesses, published by the MITRE Corporation was selected as the basis for categorizing vulnerabilities. The CWE provided us a starting point at identifying the various types of software vulnerabilities in different types of software.

In addition to the CWE, MITRE Corporation also maintains a list of standardized names for vulnerabilities and other information security exposures. This list, a community-wide effort, is known as Common Vulnerabilities and Exposures (CVE). Vulnerability names are assigned numbers. The open source systems that we picked had indexed their vulnerabilities using the Common Vulnerabilities and Exposures (CVE) nomenclature.

Open source systems studied in this project include httpd, OpenSSH, FireFox 2.0, OpenSolaris, and GNU TLS. Two open source software systems, httpd and OpenSSH, were selected to begin our analysis of vulnerabilities. The rationale for starting with these two systems was that both systems had well-documented vulnerabilities and the corresponding fixes across multiple versions. They also could be used both as standalone products and as components in many larger systems. As seen in Figure 5, we wanted to represent applications that comprised typical networks: Firefox (browser), OpenSolaris (an OS), OpenSSH (securing information), and httpd (server). Web technology was also added by evaluating Drupal. Web applications' functionality and user base has evolved along with the threat landscape. Although controls such as network firewalls are essential, they're insufficient for providing overall web application security. They provide security for underlying hosts and a means of communication, but add little to aid the application in resisting attacks against its software implementation or design. Analyzing the various applications that may comprise larger systems could reveal larger component interactions and subsequently focus aggregate system measurements.

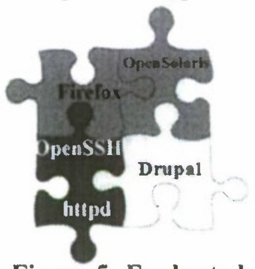


Figure 5: Evaluated Applications

The reported vulnerabilities for the versions selected (See Table 1) were accumulated and mapped to CVE vulnerability names and numbers. Table 1 contains a summary of the versions used in the analysis and the total vulnerabilities in those versions. To gain a sense of the size of the systems under study, see Table 2 for sample product versions.

Table 1: Product Versions and Number of Vulnerabilities

Product	Versions cataloged	Total Vulnerabilities
httpd	2.0.35-2.2.2	41
OpenSSH	1.2-4.3p2	27

Table 2: Total Number of Modules and Files for a Product Version

Product	Approximate No. of Modules	Number of Files
httpd 2.0.45	4972	724
OpenSSH 3.8p1	1228	243

The accumulation and identification of vulnerabilities for the subject systems take the greatest amount of the effort for this project.

The httpd reported vulnerabilities were placed into CWE categories. The primary reason for this categorization process was to understand the most common types of software vulnerabilities. This information assisted us in the process of identifying software design and code constructs that lead to software vulnerabilities. The categorization led to a better understanding of types of software vulnerabilities, a benchmark of the range of software vulnerabilities analyzed, and to finally generate a list of security metrics primitives. The vulnerabilities were mapped to the most granular classification in the CWE. To get a better top-down view of the reported httpd vulnerabilities, we coalesced all vulnerabilities into categories that were a few levels up in the classification tree. Of the 41 reported vulnerabilities, 38 were classified into CWE categories. The remaining 3 did not fit into any CWE category. All 38 of the classified vulnerabilities fell under the Location category, whereas none fell under the Motivation/Intent and Deprecated categories.

Of the 38 Location-based vulnerabilities, 37 were found in Code and 1 was found in Configuration. All 37 Code based vulnerabilities were found in the Source Code and none was found in Byte/Object Code. The categorization of the 37 source code vulnerabilities for the httpd source code is listed in Table 3.

Table 3: CWE Categorization for httpd Source Code Based Vulnerabilities

Code Quality	11
Data Handling	19
Security Features	5
Error Handling	1
API Abuse	1

The reported vulnerabilities across all versions of OpenSSH were also mapped into the most granular classification in the CWE. Of the 26 reported vulnerabilities, 20 were classified into CWE categories. The remaining 6 did not fit into any CWE category. All 20 classified vulnerabilities fell under the Location category whereas none fell under the Motivation/Intent and Deprecated categories.

All 20 Location-based vulnerabilities were found in Code and none were found in Environment and Configuration. All 20 Code based vulnerabilities were found in the Source Code and none was found in Byte/Object Code. One vulnerability was classified in two different categories, Data Handling and Code Quality. Table 4 contains the CWE categorization of the 20 source code based vulnerabilities for OpenSSH.

Table 4: CWE Categorization for OpenSSH Source Code Vulnerabilities

Code Quality	2
Data Handling	14
Security Features	2
Error Handling	1
API Abuse	2

The reported vulnerabilities across various versions of Firefox were also mapped into the most granular classification in the CWE. Of the 31 reported vulnerabilities in Firefox 2.0.0.1, 18 were classified into CWE categories. The remaining 13 did not fit into any CWE category. All 18 classified vulnerabilities fell under the Location category, whereas none fell under the Motivation/Intent and Deprecated categories.

All 18 Location-based vulnerabilities were found in Code and none were found in Environment and Configuration. All 18 Code based vulnerabilities were found in the Source Code and none were found in

Byte/Object Code. The categorization of the 18 source code vulnerabilities for the Firefox 2.0.0.1 source code is listed in Table 5.

Table 5: CWE Categorization for Firefox 2.0.0.1 Source Code Vulnerabilities

Code Quality	1
Data Handling	10
Security Features	2
Error Handling	5
API Abuse	0

Of the 35 reported vulnerabilities in Firefox 2.0.0.2, 30 were classified into CWE categories. The remaining 5 did not fit into any CWE category. All 30 classified vulnerabilities fell under the Location category, whereas none fell under the Motivation/Intent and Deprecated categories.

All 30 Location-based vulnerabilities were found in Code and none were found in Environment and Configuration. All 30 Code based vulnerabilities were found in the Source Code and none were found in Byte/Object Code. The categorization of the 30 source code vulnerabilities for the Firefox 2.0.0.2 source code is listed in Table 6.

Table 6: CWE Categorization for Firefox 2.0.0.2 Source Code Vulnerabilities

Code Quality	0
Data Handling	30
Security Features	0
Error Handling	0
API Abuse	0

Of the 92 reported vulnerabilities in Firefox 2.0.0.5, 91 were classified into CWE categories. The remaining 1 did not fit into any CWE category. All 91 classified vulnerabilities fell under the Location category, whereas none fell under the Motivation/Intent and Deprecated categories.

All 91 Location-based vulnerabilities were found in Code and none were found in Environment and Configuration. All 91 Code based vulnerabilities were found in the Source Code and none were found in Byte/Object Code. The categorization of the 91 source code vulnerabilities for the Firefox 2.0.0.5 source code is listed in Table 7.

Table 7: CWE Categorization for Firefox 2.0.0.5 Source Code Vulnerabilities

Code Quality	2
Data Handling	69
Security Features	20
Error Handling	0
API Abuse	0

We collected similar data and performed similar analysis on OpenSolaris. All 24 reported vulnerabilities were classified into CWE categories. Out of the 24 classified vulnerabilities, all 24 fell under the Location category whereas none fell under the Motivation/Intent category.

Out of the 24 Location based vulnerabilities, all 24 were found in Code and none were found in the Configuration. Out of the 24 Code based vulnerabilities, all 24 were found in the Source Code and none were found in Byte/Object Code. The categorization of the 24 source code vulnerabilities for the OpenSolaris source code is listed in Table 8.

Table 8: CWE Categorization for OpenSolaris Source Code Vulnerabilities

Code Quality	6
Data Handling	5
Security Features	8
Time and State	5
Error Handling	0
API Abuse	0

It is obvious that the majority of the reported vulnerabilities are in the source code. Since source code is a descendant of the software design, it seems intuitive that errors in design may have caused the majority of the vulnerabilities identified.

7.2 Identifying Vulnerable Modules

Vulnerabilities were cataloged for all versions of the selected open source software. Many references were consulted to identify the exact nature and location of each reported vulnerability. For example, vulnerability reports for httpd were collected from the Apache project's website [Apa07]. OpenSSH did not keep detailed vulnerability reports for the portable branch, so these were collected from Security Focus [Sec07]. When necessary, information from the MITRE Common Vulnerabilities and Exposures (CVE) database [MITR07], bug databases, and mailing lists were used. Because these systems were written in C, there was some difficulty in determining module granularity. Because there is no concept of classes or namespaces, it was decided that module granularity should be at the function level, and the terms \module" and \function" can be used interchangeably. Many of the vulnerabilities are documented in the MITRE CVE database. For each vulnerability, the corresponding CVE number (if provided) and a possible CWE classification were recorded. These were to help catalog the type of vulnerability. The affected versions of the software were also cataloged. Affected versions were not always accurately provided for the OpenSSH vulnerabilities, so any uncertainties were noted. In order to analyze the vulnerabilities, each one was narrowed down to the file and function in which it originated. Sometimes location information was obtained by looking at published patches to the code. When published patches were not available, fixed versions and vulnerable versions of the code were run through diff, and the resulting output was carefully analyzed. Developer mailing lists were useful in narrowing down the vulnerability locations. Once located, the nature and specific causes of the vulnerabilities were noted.

There are a few limitations in cataloging the exact locations of the vulnerabilities within the code. In cases where the location was not published, diff output had to be analyzed to find the modified functions. Since we are not familiar with the code, it is possible that some functions may have been mistakenly labeled as vulnerable or at least contributing to the vulnerability in some way. Another difficulty in finding exact locations within the code is that security vulnerabilities often arise as a result of communication between modules, rather than the modules themselves. For example, if the responsibility for filtering input is split between a few modules, and part of a string is accidentally left unaltered, it is uncertain which module was responsible. Because of such situations, any potential participants in a vulnerability were documented as well. Figure 6 displays, in general, our process of extracting the information required to validate vulnerability predictiveness. The beginning tasks can be divided into two parallel processes, one analyzing the selected system vulnerabilities and identifying the modified code, and the other collecting the metrics on the selected system. Once these two tasks are performed the information is matched and merged to form the data for vulnerability analysis.

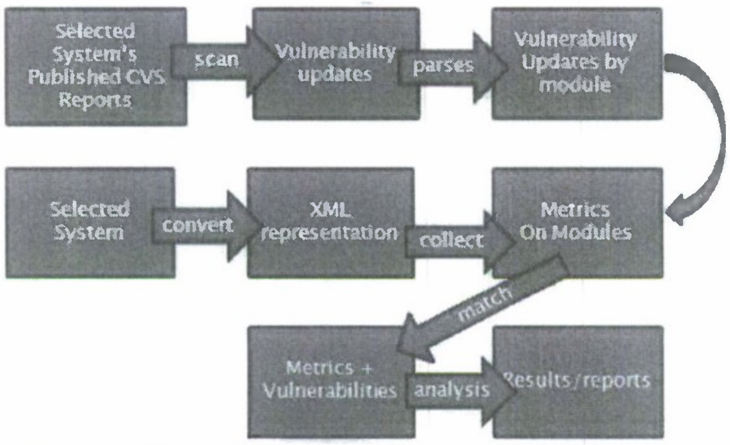


Figure 6: SMART's Data Collection and Analysis Process

After collecting the vulnerability reports, a few patterns were noted. Vulnerabilities were often found in the following areas:

1. Large functions.
2. Areas of high complexity.
3. Areas of input handling.
4. Areas involving encryption protocols.
5. Areas involving platform-dependent code.
6. Asynchronous or concurrently called functions.

In addition to this list, there were many vulnerabilities that didn't seem to fit into any of those categories. These observations, along with the CWE categories, were used as a general guideline of what to look for when analyzing the code. If some properties of these observations can be statically measured, then it is possible that these may be used as primitives for a future metric.

7.3 Metrics to Identify Vulnerabilities

A series of metrics were used to attempt to capture information regarding the environment and internal properties of a module. In particular, many of these metrics were geared at pinpointing the previously mentioned observations. Quantitative metrics were computed for Apache httpd versions 2.0.36 and 2.0.48, and OpenSSH 3.2.2p1, 3.7.1p1, and 3.8p1. These specific versions were chosen because collectively they were representative of the types of vulnerabilities suffered by each software package.

A commercial tool, C++ Understand, was selected to assist in the collection of other metrics. This tool also includes an API to customize the collection process. The tool collects various metrics such as cyclomatic complexity, nesting levels, knots, and in/out-flows at the function scope, as well as various metrics at the file and project scopes. With C as the source language, a module was restricted to that of a function definition. In addition to the metrics provided by the tool, other customized metrics were collected based on our previous list of observations.

We hypothesized that areas of high I/O and asynchronous events may be found by finding the appropriate system calls in the code. However, functions doing raw I/O in these software packages are not usually vulnerable to attack. It was thought that because these functions would play a large part in the attack surface, they would possess more vulnerabilities. Instead, vulnerabilities were found further up the call tree, where complex logic was applied to the input.

A metric for pinpointing areas addressed in our initial observations was desired. We felt that scanning for relative positions of I/O and system calls would be helpful. Originally, a call tree metric was developed that counted the number of levels in the call tree above an I/O call. It is very possible that the location of a module relative to an I/O call may serve as a metric primitive. However, unexplained inaccuracies were discovered in the tool output. It is uncertain whether these inaccuracies were due to a flaw in the tool, or a limitation of the Understand for C++ API. Because of the inaccuracies, this metric was omitted.

As an alternative, a neighborhood metric was developed. In the neighborhood metric, a module is scanned to see how "close" it is to a vulnerability. Given a module m, a call tree depth d and a list of vulnerable modules v, the neighborhood metric recurses d levels in the call tree above and below m, and counts the number of modules from v that were encountered. Such a metric may identify environmental properties of a vulnerability.

To account for code complexity, the D_e, D_i, and D(G) metrics were used. These metrics have been shown to identify complexities in code with much success [ZAGE90, ZAGE93]. These design metrics also identify areas that may need to be redesigned. This is very useful when trying to identify design factors leading to security flaws.

Metrics collected by C++ Understand were combined to form composite metrics using the definitions of D_e, D_i and D(G) as published in [ZAGE93]. It was first assumed that the conventions that are used in C++ Understand to collect the primitives for these design metrics would result in design metric values that closely approximate those calculated by the Zages and validated over a 17-year period. This was not the case. In what follows, we will refer to the design metrics from C++ Understand, in *italics* as D_e, D_i and *D(G)*.

Most of the open source applications selected consisted of C files. Only Drupal consists of PHP, INC, JavaScript, Perl, XML files. To many, PHP is one of the most popular and wide-spread web languages. For each analysis, the source files are converted into a customized XML representation. For the C source files, there is the Design Metrics Analyzer (DMA) tool. For PHP we identified a tool, PHP-AST that parses a PHP document to create an XML document. The XML output is highly desired because the DMA tool uses XML as the fundamental input to collect metrics. After several investigations, it appears that PHP-AST lists the statements internal to a PHP module as an xml tag label ACTION*xx*, where *xx* is an integer counter. At first, it looked as if each statement type was mapped to a particular ACTION, but this does not appear to be case. They are assigned and incremented as the application parses through the PHP modules. A defined action to tag is required for our analysis, thus the development of a PHP2XML was initiated.

The C open source applications were reevaluated. The metrics were recalculated for Apache v1.3.1. Version 1.3.1 consists of 144 files with an identified 8 vulnerabilities. For the D_e metric analysis, with 10% of the modules highlighted, **87.5% or 7 of the 8 vulnerable modules were identified**.

The analysis of OpenSolaris began. OpenSolaris has 22,600 files in the downloadable tar file. An attempt was made analyze all the files. DMA ran out of memory at approximately the 5000th file. These files totaled several million LOC. From this fast amount of code, twenty-three modules had thirty-seven changes because of vulnerabilities. To identify the vulnerabilities, we are trying to select one out of every 10,000 modules! The vulnerability updates were grouped into two stems of the source code, eighteen modules on one stem and five modules on the other for the total of twenty-three modules. The "5-stem" consisted of twenty-nine files or 90,417 xml tags. The metrics were calculated for this stem. For the D_e metrics analysis, with 10% of the modules highlighted, **60%, three out the five vulnerable modules and 69%, nine out of thirteen changes were identified.**

The C/C++ programming language is incomplete without its macro preprocessor. With disciplined use the preprocessor can reduce programmer effort, improve portability and performance. It is also viewed as a source of difficulty for understanding, modifying and analyzing C/C++ programs. C++ is a terrible language for tool vendors to handle. There are only a handful of people in the world capable of writing an accurate parser to read and understand C++ source files in all their template-riddled complexity. Additionally, the preprocessor transforms the visible source code into something quite different before it is submitted to the compiler. Unless the exact set of preprocessor defines are known to the static analysis tool, an overwhelming number spurious or plain incorrect warnings can be expected or just silenced. Most tools make no attempt to analyze macro usage, but simply preprocess their input. Since the type of applications that are being analyzed for vulnerabilities are cross-platform, many of the C source files are riddled with preprocessor code. This may be confounding difference between tools that analyze C/C++.

We have evidence that suggests design plays an important role in software security, since many of the vulnerable module design metrics primitives would be stress points in the subject system. We have also isolated calls within modules that use, access or modify system resources, such as

- Channels – sockets, rpc
- Services
- Executables - libraries
- Files – config, log, temp, authorization, metadata
- Scripts
- Accounts
- Symbolic link
- CGI
- Web pages
- Any persistent data

7.3.1 Comparisons of Design and Structure

In order to investigate the impact of design on security, it is necessary to comment on the designs of some of the selected software. Both OpenSSH and httpd are portable software, meaning they execute on many different architectures and operating systems. OpenSSH is restricted to Unix variants. In addition, each software suite has multiple features that can be switched on and off at compilation time or modified at run-time. Both software packages are written in C. The two systems have varying approaches for accomplishing portability. OpenSSH has two development branches, an OpenBSD branch and a portable branch. The portable branch is taken from the OpenBSD specific branch and modified to run on other supported operating systems. As a result, the design of the system is not changed for the portable branch. Instead, preprocessor directives are inserted into the existing code to control compilation of platform-specific features. This may cause code to be longer and more confusing, and possibly add complexity.

Apache httpd is a modular system, meaning major features are implemented as separate modules. Unlike OpenSSH, portability is accomplished by having different implementations of commonly-used abstractions for each supported operating system. This allows for clean separation of portable and non-portable code, rather than mixing portable and non-portable code as in OpenSSH. The web server itself sits atop the Apache Portable Runtime (APR) library. The APR library provides an abstraction layer for commonly used operations such as memory allocation, file I/O, input sanitizing, etc. These features aid in porting httpd to new platforms. For systems written in C/C++, it appears that the use of preprocessor constructs and libraries increase the probability of vulnerabilities.

7.3.2 Design Metric D(G) Analysis

D(G) values were computed for each module to gauge the effectiveness of this design metric at identifying vulnerable areas of the systems. The results are shown in Table 9. Many of the vulnerable modules possessed high D(G) values, suggesting once again that design plays a significant role in secure software. Relationships and differences between modules with high and low D(G) values need further exploration. A technique, which we call *nearest neighbor*, where the neighbors of vulnerable modules are explored further for identifying characteristics such as a high design metric count was implemented.

Table 9: Effectiveness of the C++ Understand D(G) in Identifying Vulnerabilities

	Apache 2.0.36	OpenSSH 3.2.2p1	OpenSSH 3.7.1p1	OpenSSH 3.8p1
Total Vulnerable Functions	35	28	33	35
# Found in top 10% of D(G) values	24	12	13	13
# Found in top 25% of D(G) values	29	18	21	22

The design metrics analysis showed some interesting results. It was able to identify the majority of the vulnerabilities in Apache httpd. This strongly suggests that httpd has some large modules that need to be refactored. However, these same results were not reflected in OpenSSH. D(G) was more effective with Apache, perhaps due to the design of httpd, particularly the abstractions in the APR library (which we did not analyze). Many of OpenSSH vulnerabilities were in smaller functions with low complexity. Apache vulnerabilities were in larger, more complex modules. In httpd, most of the utility functions are in the APR abstractions, where the functions tended to be small and specific in their tasks. It is possible that the heavy use and testing of the APR may have abstracted out many potential vulnerabilities, leaving the remainder of the vulnerabilities in the larger, more complex areas.

Many vulnerabilities in the code had high D_e and data structure manipulation (DSM) values. (DSMs are one primitive metric in the composite metric D_i.)As shown by the design metrics data, high coupling itself is not a factor for vulnerability prediction since there are many modules in these systems with high D_e and DSM values that are not vulnerable. However, this does not imply that coupling is not relevant in the security of a module. Good abstraction may reduce the impact a vulnerability has on the rest of the system. Reducing coupling and keeping D_e and DSM values low may be a good preventative measure during the design phase of a software application.

7.3.3 Qualitative Observations

The vulnerabilities in the systems can be divided into vulnerabilities which span multiple modules and those contained in single modules. No measurable commonalities and differences could be made between multiple- and single-module vulnerabilities. However, some common traits were noticed within each category.

Multi-function vulnerabilities often fell into two categories. In one category were the repetitive vulnerabilities. In these cases, either the same mistake was copied across several modules (possibly due to copy/paste operations by the developer), or an API call was changed and multiple modules needed to be updated (such as the addition or removal of a return value that needed handling). In the other category were vulnerabilities which resulted from failures in module cooperation.

This first category tended to include the vulnerabilities resulting from coding errors such as buffer overflows, incorrect return values, and other simple mistakes in coding. These often required simple fixes to an individual module. However, there were cases in which a module needed to change its return value, and this required changes to propagate to all modules using it.

The second category is interesting because these sometimes require refactoring to fix the issue. For other vulnerabilities in this category, the fix is to modify a few different modules to fix an unhandled case.
One example of these kinds of vulnerabilities is a function that allocates a resource, and the responsibility to dispose of the resource is not explicitly given to another module (or the other module fails to clean it up, such as during the handling of an error). Therefore, the resource is leaked. A similar situation has occurred in the Apache httpd thread pool code. In this case, an abstraction had to be added to fix some function cooperation issues resulting in threading vulnerabilities. This category of multi-module vulnerabilities tended to include some of the more non-trivial fixes. Some vulnerabilities were also caused by failure to clean up resources during the handling of an error.

7.4 Statistical Analysis

Various statistical tests using the statistical tool, SPSS, were run on the study data. Each data set consisted of metrics data for each function from the system and a Boolean indicating whether or not the particular module had a reported vulnerability. Since the outcome for our data set is dichotomous (a module is vulnerable or not vulnerable), binary logistic regression was used. For a logistic regression, the predicted dependent variable is a function of the probability that a particular module will be in one of the categories (for example, the probability that module x is vulnerable, given the module's metric value for the set of predictor metrics). The results of the logistic regression can be used to classify modules as vulnerable or non-vulnerable. Our best results have allowed us to correctly classify 88% of the modules.

The situation we encountered during the initial logistic regression runs is that the number of vulnerable modules is statistically insignificant compared with the number of non-vulnerable modules of the system. With the assistance of Dr. Darius Lecointe, a Ball State University statistician, the decision to perform logistic regression analyses using a number of randomly selected non-vulnerable modules equal to the number of vulnerable modules within each project provided improved results.

For each project multiple runs were initiated. The results varied depending on the modules chosen randomly by SPSS to include in the analysis. The httpd analyses commonly included complexity metrics. The openSSH models were not that consistent. Data from a version of httpd and data from one version of openSSH were combined and a logistic regression was run on the combined data sets. Consistently in the models, the number of inputs and nesting levels, both indicators of complexity, were present in the models.

The top thirty functions with the highest number of vulnerabilities in their first-level neighborhoods were taken for each system. What is interesting about the data is that it confirms the previously mentioned initial observations. Across all of the projects, string/buffer operations, I/O and input filtering functions, and some thread and mutex-related abstractions were located in the lists. Since vulnerable functions were using these, coupling with these kinds of functions may serve as a future environmental metric. The number of times a given function couples with an item in these lists may be a valuable metric as well.

7.5 Operational Thread Based Reliability Estimation

The reliability of large scale computerized networks dispersed over widely scattered geographical areas is difficult to determine, particularly when the networks include hardware, software and human elements as well as the complex communications links between nodes of the network. Further, to estimate the reliability of such a network during the design phase of development or the effect of inserting a new network node is even more difficult. These types of networks are not uncommon in international business, industry, government and military operations, and therefore estimates of their reliability are important to users.

A common foundation for uniting these diverse and scattered network elements can be based on the concept of the operational thread. An "operational thread" is a term used by the U.S. military services as a series of related operational tasks that are specifically focused to highlight the contribution of experimental initiatives or infrastructure systems for basic Command and Control (C2) process. This term can be generalized for tasks involving multiple services/components composed of hardware, software and human elements as well as communications links. Figure 7 depicts three hypothetical operational threads each completing a C2 process.

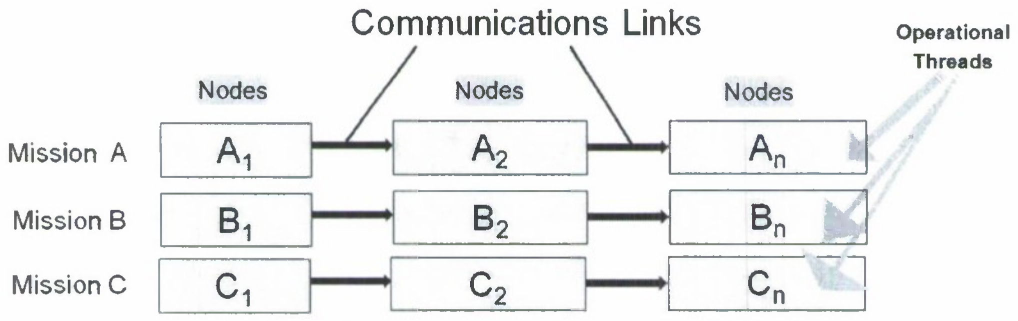


Figure 7: Three Hypothetical Operational Threads

Each mission can cut through various components of a service-oriented architecture. Each node in Figure 7 can be comprised of several software, hardware and human elements in any order. Figure 8 depicts one operational thread with three nodes (components), each containing different thread element types. Each node can be geographically disbursed and can be connected by different types of communication links, identified in Figure 8 as $Link_{12}$ and Link $_{23.}$ To compute the reliability of an operational thread, one must calculate the reliability of each of the individual components comprising the node as well as the reliability of the connecting links ($RL_{i,i+1}$). Thus the estimated reliability for component 1, RC_1 in Figure 8 is

$$RC_1 = R_{hw} * R_{sw} * R_{hw}$$

The estimated reliability for component 2 (RC_2) and for component 3 (RC_3) in Figure 7 is calculated in the same manner.

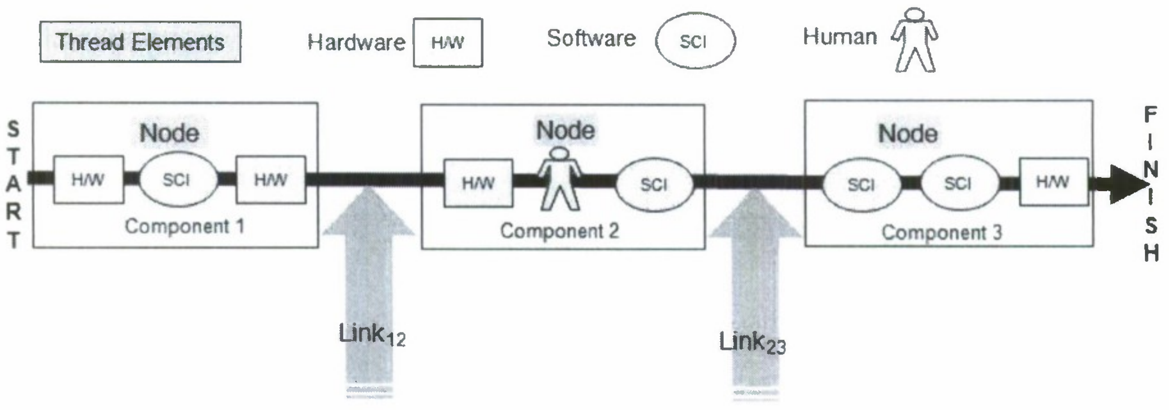


Figure 8: An Operational Thread with Three Components (Nodes)

The reliability of hardware is a well-researched field and numerous useful approaches exist. The reliability estimation for software and the human element contained in the equations is not as easy to calculate. At the design phase of the system, node or component, without previous history, these estimates using current reliability techniques are impossible to estimate with any degree of accuracy. However, through the extension of the design metrics technology, a basis for a software reliability estimate has been developed. This technique can provide an estimate of the reliability of the software component only using current design artifacts or code. In purely hardware and software operational threads, the missing part of the reliability equation is the link. At this point, the human element will be excluded from the analysis. Communication reliability estimates are again a well-researched field. Depending on the type of communication, standard tools are available to estimate the reliability of these links, for example, NIST's Link Budget Calculation. Now possessing an estimate for each of the components and for the links, the reliability of an operational thread, R_{ot} can be calculated as

$$R_{ot} = RC_1 * RL_{12} * RC_2 * RL_{23} * \ldots * RC_m * RL_{mn} * RC_n.$$

Finally, working upward from the reliability of operational threads, the estimated overall reliability of the network itself is derived.

7.6 Extending a Reliability Model

A software reliability model based on the design metrics D_e and D_i has been developed. We suspect that reliability and security problems have significant software weaknesses in common. For example, data corruption is an area where reliability and security have the same underlying technical problem with slightly different emphases. Reliability can be affected through the accidental corruption of data whereas security focuses on the malicious attacks to corrupt data. According to a preliminary analysis done by the SEI's CERT® group, over 90% of software security vulnerabilities are caused by known software defect types. The analysis also showed that most software vulnerabilities arise from common causes: the top ten causes account for about 75% of all vulnerabilities. Another analysis from At Stake Research of forty-five e-business applications showed that 70% of the security defects were software design defects.

7.7 Testing, Vulnerabilities and Metrics

To improve vulnerability detection, frequent testing and measurement will be required. In most traditional development processes the norm is poor metrics and poor visibility into the process. Testing itself provides a level of confidence and visibility into the process and product. We also investigated the effectiveness of model-based test generation techniques in the detection of errors in implementations of security protocols. An experiment using GNU TLS as the product under test and generating test cases manually from the state chart using the test tree found in the well-known W-method was initiated. The result of this experiment was that 19 to 49% of the code in various components remained untested within the product. The coverage was measured using the FAA mandated MC/DC coverage criterion. The analysis also revealed that faults in the untested portions could lead to improper authentication, denial of service, and loss of confidentiality. For model-based testing using state charts derived from requirements, a simple recommendation is that this process must be augmented with coverage analysis. However, what is a safe amount of coverage? Combining test coverage and component metrics may provide the obvious answer. Component metrics and coverage will be analyzed simultaneously to determine patterns that allow developers and project managers to focus their testing and inspection efforts to obtain the minimum coverage necessary while testing the fault-prone modules.

7.8 SMART Model

Many security vulnerabilities are the result of defects unintentionally introduced during the development process. To significantly reduce software vulnerabilities, overall defects must be reduced. It is intuitive that defective software is seldom secure. For this reason, the SMART project combines the vulnerability analysis based on the outcomes of the current studies, our reliability model based on the design metrics and the process information derived from testing for the SMART model. Overall defect reduction, directed vulnerability identification and process information is the key to secure software.

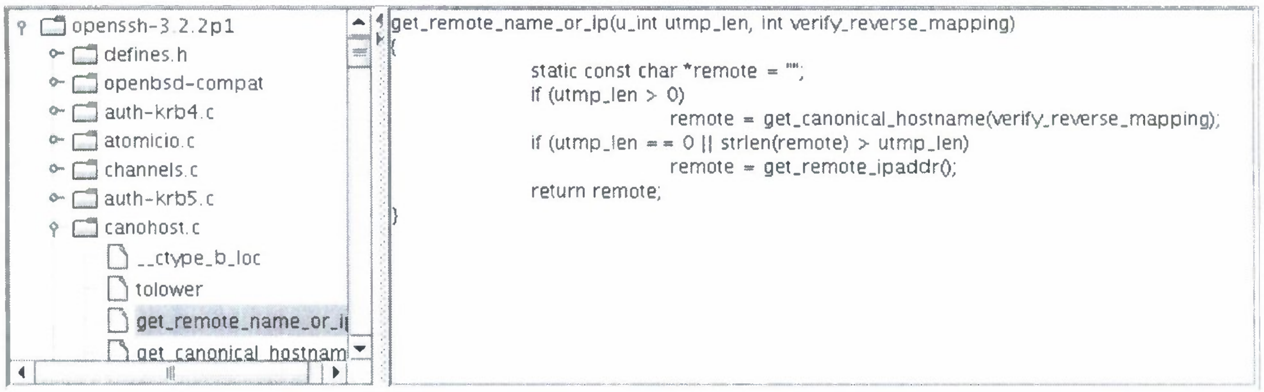
7.9 Visualization

As mentioned earlier. *RAVE* is the **R**eliability **A**ppraisal and **V**ulnerability **E**valuation tool. It is the software that incorporates our research methodology. There are two major components to RAVE, referenced simply as the front-end and back-end. The back-end is used to analyze program source code and compute various design metrics. The front-end is the user-interface to the back-end.

Developing the RAVE front-end involves the integration of several tools and technologies. An experimental, proof-of-concept system was developed that verified the feasibility and applicability of a three-screen methodology. This system demonstrated that three screens could be used effectively to explore a software system by dedicating views to program source code, software visualization, and metrics analysis presentations.

The RAVE front-end uses the data and visualization architecture of prefuse, which is available under a BSD-style license. The overall software design applies the Visual Proxy architectural pattern, a specialization of the Presentation-Abstraction-Control pattern. This architecture facilitates the development of UI "plug-ins". The RAVE front end, then, is a shell that loads these plug-ins, of which there are currently three, one for each view of the software.

The static display of program source code is straightforward. RAVE adopts a common tree-based model, treating directories and files as non-leaf nodes and functions as leaf nodes of a tree as seen in Figure 9.



```
get_remote_name_or_ip(u_int utmp_len, int verify_reverse_mapping)
{
		static const char *remote = "";
		if (utmp_len > 0)
				remote = get_canonical_hostname(verify_reverse_mapping);
		if (utmp_len == 0 || strlen(remote) > utmp_len)
				remote = get_remote_ipaddr();
		return remote;
}
```

Figure 9: RAVE Interface for Displaying Selected Program Source Code

Conversely, developing usable and appropriate visualizations is a significant research problem. The requirement for these visualizations is that they clarify metrics-based software vulnerabilities. Based on our research findings thus far as well as concurrent research activities, we have adopted a visualization technique the highlights neighborhoods of functions. Regional call graphs are shown in 10 using a radial layout, and up to three levels of depth are currently supported.

Displaying three or more levels of depth has reduced the usability of these graphs in our experimentation. We have also experimented with clustering, i.e., force-directed drawings of an entire software system. However, for our experimental data, specifically OpenSSH, the result is too cluttered to be usable.

The metrics table information is gathered from the metrics collector specified to RAVE. This back-end can be replaced with any system that follows the protocol we have defined, and the current implementation relies upon the STI Understand packages. This allows for the collection of dozens of metrics, including McCabe and Zage metrics. A security engineer can select which metrics to compute, and these are all shown in the table. By sorting on various columns, the engineer can use the metrics to determine which modules to highlight as vulnerable. For example, one may wish to compare cyclomatic complexity against D(G), marking the highest 10% of each as potentially vulnerable. These vulnerabilities are then highlighted in the visualization as well, providing the user with an intuition for how the vulnerable modules are related.

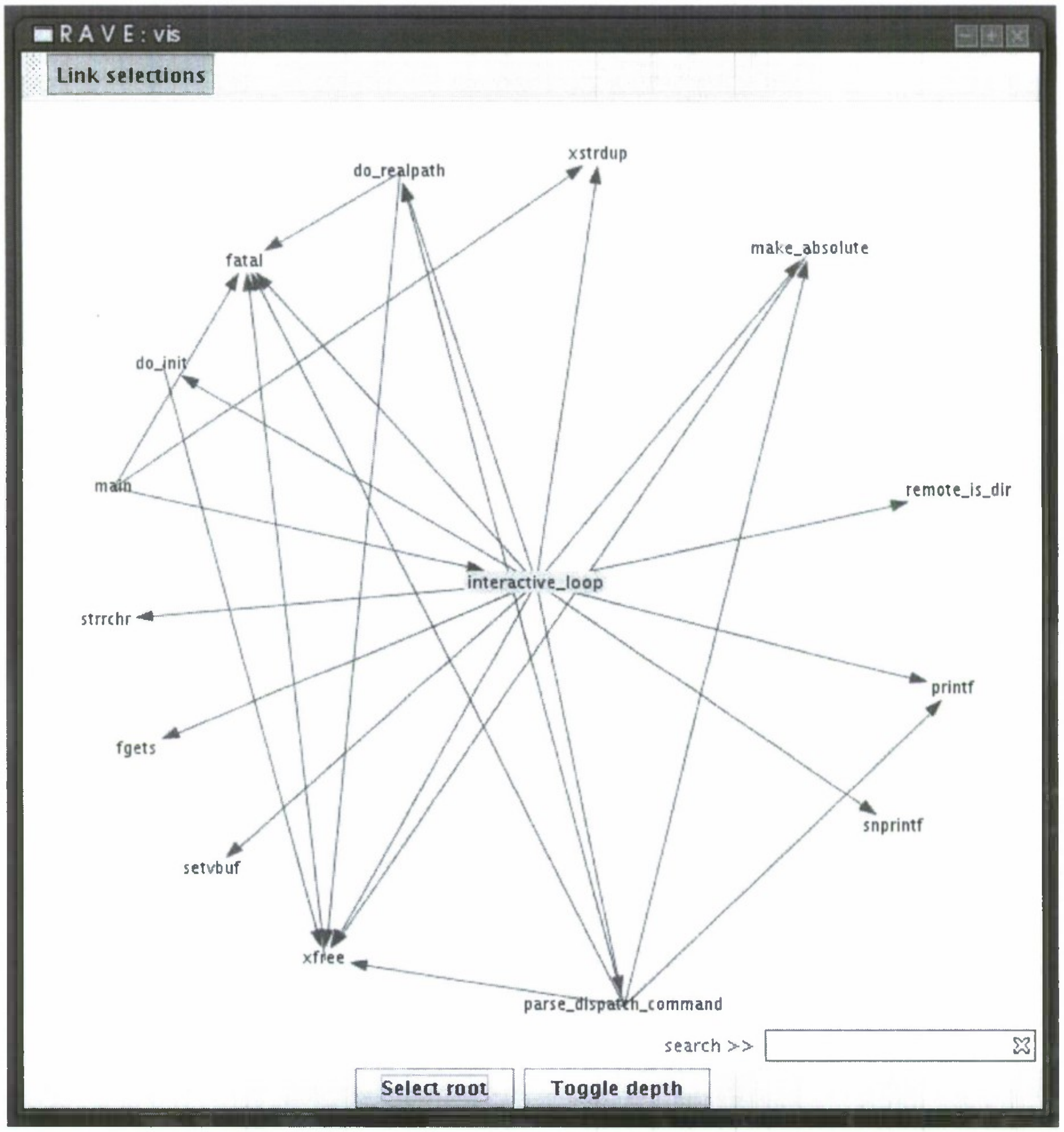


Figure 10: RAVE Interface Displaying Regional Call Graphs Using a Radial Lay

8. Conclusions

Many of the modules with vulnerabilities in the open source systems studied had high D(G) values. While D(G) may not be a predictor for the degree of security of a module, the fact that many of the vulnerabilities in the systems had high D(G) values suggests that design plays a significant role in the robustness and security of software. It seems that there is no clear way to detect vulnerabilities based solely on design metrics. However, complex problems, such as security, often need more data and concentrating on a single aspect is not likely to have a high payoff. In our design metrics work, we assess both the external and internal complexity of a module. A module's complexity can stem from either internal factors, or external factors or both. We believe that assessing a module's vulnerability also will require more information, since each development is essentially unique, created by new people, with different environments, for new applications, and more importantly, is dependent on the system's environment and other interplaying applications. For these reasons, we have identified enhancements to the suite of design metrics to capture potential environmental factors and the abstractions around data structures. These need to be explored further to form the basis for the E_e and E_i metrics. For example, DSM values are important, but it may be beneficial to combine DSM counts with a measure of cross-module coupling to determine how much of a structure is shared. An overall data structure abstraction metric would be a useful addition. Neighborhood analysis enhancements may lead to a better understanding of vulnerability locality. Extending the neighborhood analysis to include data structures could prove to be useful. Since security structures may be shared by multiple modules and distributed throughout the application, neighborhood analysis could provide the insight into vulnerabilities that span multiple modules. Some integration with the attack surface concept may also enhance the design metrics, such as increasing the weights on I/O counts. Enhancing D_i with a system call count may increase vulnerability detection where software interacts with the operating system. With further research and development of the design metrics, the essence of the design metric qualitative observations made earlier may carry over to the security domain.

Until then, based on our observations and the past work by others, the following suggestions may prove helpful:

- Provide abstraction around shared and local data structures to avoid unnecessary coupling.
- Design the abstractions in such a way as to control and restrict direct access and manipulation of the data. Keep them as close to "black boxes" as possible. It may be necessary to use immutable objects and data structures.
- Keep all abstractions and procedures in the system small, simple, and very specific in purpose. Make sure their semantics are well-defined.
- Preserve referential transparency by eliminating potential side-effects when calling functions.
- Make all interfaces and API's easy to use correctly and difficult to use incorrectly.
- Whenever possible, modules should clean up and dispose of all resources they allocate. When this is not possible, data and resources returned by functions should be returned as abstract types.

Researchers have offered tools to detect certain vulnerabilities. Some have suggested languages or paradigms to solve the problems of software vulnerabilities. Security test suites have also been offered to help reduce the threat. However, decisions made at design time still have a large impact on the overall security of a system, not to mention the reliability and maintenance implications.

9. Dependencies

Our project was dependent upon the availability of systems with process data, including defect reports and vulnerability assessments.

10. Contacts at ARL

Our primary contact is Glenn Racine, Chief, Battlefield Communication Networks Branch, Army Research Laboratory, Adelphi, MD 20783-1197.

11. References

[BUCH00] Adam L. Buchsbaum and Jefferey R. Westbrook. Maintaining hierarchical graph views. In *SODA '00: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages

566–575, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics. ISBN 0-89871-453-2.

[BRYA05] E. Bryant, J. E. Davis, D. Zage and W. Zage, "CATS^2 - Cost of Anti-Tamper Software Security", *Proceedings of the Anti-Tamper 2005 Conference,* Sandia National Labs, April 19-21, 2005.

[CHAO93] A. Chao, M. C.-Ma and M. C. K. Yang, "Estimation and stopping rules for recapture debugging with unequal detection rates," *Biometrika*, vol. 80, No. 1, pp. 193-201, March 1993.

[CHEN94a] M. H. Chen, "Tools and Techniques for Testing-Based Software Reliability Estimation," Doctoral dissertation, Department of Computer Science, Purdue University, W. Lafayette, IN, April 1994.

[CHEN94b] M. H. Chen, A. P. Mathur, V. J. Rego, "A Case Study to Investigate Sensitivity of Reliability Estimates to Errors in the Operational Profile," *Proceedings of the Fifth International Symposium on Software Reliability Engineering,* IEEE Computer Society Press, Monterey, California, November 6-9, 1994, pp. 276-281.

[FLOC95] J. Floch,, "Supporting Evolution and Maintenance by Using a Flexible Automatic Code Generator", *Proceedings of the International Conference on Software Engineering* '95, Seattle, WA, 1995.

[GOLU05] C. Golubitsky, "Toward an Automated Vulnerability Comparison of Open Source IMAP Servers", *Proceedings of the 19th Large Installation System Administration Conference (LISA05)*, 9-22, San Diego, CA, 2005

[HEFF04] J. Heffley and Pascal Meunier, "Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security?", *Proceedings of the 37th Hawaii International Conference on System Sciences,* 2004.

[HOWA03] M. Howard, J. Pincus and J. Wing, Measuring Relative Attack Surfaces, Proceedings of Workshop on Advanced Developments in Software and Systems Security (2003).

[HORG95] J. R. Horgan, A. P. Mathur, A. Pasquini, and V. FJ. Rego, "Perils of Software Reliability Modeling," Technical Report, SERC-TR-P-160, February, 1995.

[LANZ03] M. Lanza, "Program Visualization Support for Highly Iterative Development Environments", *Proceedings of the Second IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT '03)*, Amsterdam, The Netherlands, September 2003.

[MANA05] P. Manadhata and J.M. Wing, "An Attack Surface Metric," CMU-CS-05-155, Technical Report, July 2005.

[MITR07] MITRE. Common Vulnerabilities and Exposures. http://cve.mitre.org, 2007.

[MURD06] J. Murdoch, "Security Measurement". White paper, Practical Software and Systems Measurement, v3.0, January 2006.

[MUSA87] J. D. Musa, A. Iannano, and K. Okumoto, Software Reliability, Measurement, Prediction, Application, McGraw Hill Book Company, New York, 1987.

[NOEL04] Steven Noel and Sushil Jajodia. Managing attack graph complexity through visual hierarchical aggregation. In *VizSEC/DMSEC '04: Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, pages 109–118, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-974-8. doi: http://doi.acm.org/10.1145/1029208.1029225.

[SHEY02] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M. Wing. Automated generation and analysis of attack graphs. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 273, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1543-6.

[STIN05] J. Stineburg, W. Zage and D. Zage, "Measuring the Effect of Design Decisions on Software Reliability", International Society of Software Reliability Engineers (ISSRE) 2005 Conference, Chicago, November 2005.

[WONG00] E. Wong, W. Zage, D. Zage, J.R. Horgan (Telcordia Technologies) and M. Syring (Motorola), "Applying Design Metrics to Predict Fault Proneness: A Case Study on a Large Scale Software System," *Software-Practice and Experience*, Vol. 30, No. 14, pp. 1587-1608, November 25, 2000.

[YANG95] M. C. K. Yang and A. Chao, "Comparisons of methods in reliability estimation and stopping rules for software testing," *IEEE Transactions on Reliability*, vol. 44, pp. 315-321, 1995.

[ZAGE90] W.M. Zage and D.M. Zage, "Relating Design Metrics to Software Quality: Some Empirical Results", SERC-TR-74-P, May 1990.

[ZAGE93] W.M. Zage and D.M. Zage, "Evaluating Design Metrics on Large-Scale Software", *IEEE Software,* Vol. 10, No. 4, July 1993.

[ZAGE99] W.M. Zage, D.M. Zage, J.M. McGrew, and N. Sood, "Using Design Metrics to Identify Stress Points in SDL Designs", *Proceedings of the 9th International SDL Forum,* Montreal, Canada, published by Elsevier Science, UK, June 1999.

[ZAGE03] W.M. Zage and D. Zage, "An Analysis of the Fault Correction Process in a Large-Scale SDL Production Model," *Proceedings of the 25th International Conference on Software Engineering*, Portland, OR, May 2003.

[ZAGE06] W.M. Zage and D.M. Zage, "Metrics Directed Verification of UML Designs", SERC Technical Report 284, June 2006.

BALL STATE UNIVERSITY

Business Affairs
Contracts and Grants Office

Muncie, Indiana 47306-0750
Phone: 765-285-5221
Fax: 765-285-5224

November 25, 2009

Defense Technical Information Center (DTIC)
8725 John J. Kingman Road, Suite 0944
Ft. Belvoir, VA 22060-6218

SUBJECT: Grant Number: W911NF-06-2-0030
Title: 2006-09 DOD/US Army Software Reliability & Security
Period of Award: 07/01/06 – 08/31/09
Award: $999,000.00
Principal Investigator: Wayne Zage
BSU No.: 5-46127

Enclosed, please find the Final Report for the above mentioned grant.

If you have any questions, please contact me at 765-285-5288, or by e-mail at estrauch@bsu.edu.

Thank you.

Sincerely,

[illegible]

Eric D. Strauch
Grant Specialist

Enclosures